

Resource control of object-oriented programs

Jean-Yves Marion and Romain Péchoux

Loria, Carte project, B.P. 239, 54506 Vandœuvre-lès-Nancy Cedex, France,
and École Nationale Supérieure des Mines de Nancy, INPL, France.

`jean-yves.marion@loria.fr` `romain.pechoux@loria.fr`

February 1, 2008

Abstract

A sup-interpretation is a tool which provides an upper bound on the size of a value computed by some symbol of a program. Sup-interpretations have shown their interest to deal with the complexity of first order functional programs. For instance, they allow to characterize all the functions bitwise computable in **Alogtime**. This paper is an attempt to adapt the framework of sup-interpretations to a fragment of oriented-object programs, including distinct encodings of numbers through the use of constructor symbols, loop and while constructs and non recursive methods with side effects. We give a criterion, called brotherly criterion, which ensures that each brotherly program computes objects whose size is polynomially bounded by the inputs sizes.

1 Introduction

A sup-interpretation is a tool introduced in [13] in order to deal with the Implicit Computational Complexity of first order functional programs. Basically, the sup-interpretation of a first order functional program provides upper bounds on the size of any value computed by some function symbols of the program. The notion of sup-interpretation is a descendant of the notion of quasi-interpretation. See [5] for a survey on quasi-interpretations. It has been demonstrated in [12], that the notion of polynomial sup-interpretation strictly generalizes the notion of polynomial quasi-interpretation. In other words, every polynomial quasi-interpretation is a polynomial sup-interpretation and there are programs which admit a sup-interpretation and no quasi-interpretation. As a consequence, sup-interpretation provides more intentionality than quasi-interpretation, i.e. it allows to capture the complexity of more algorithms. Such a flexibility is very interesting when we consider small complexity classes. For example, in [6], sup-interpretations allow to characterize all the functions bitwise computable in alternating logarithmic

time. Another interesting consequence developed in [13] consists in an application of the sup-interpretation tool to termination criteria such as the dependency pairs [3] or the size change principle [11].

The notion of quasi-interpretation has already been extended to Byte-code verification and to reactive programs. See for example [1, 2, 7]. Consequently, a major issue consists in the adaptation of the sup-interpretation analysis to imperative and object-oriented programs. We try to tackle this problem in this paper by enlarging the framework of sup-interpretations to a fragment of object-oriented programs without recursion. Our language is very similar to the language studied in [9]. However, since we consider assignments, it is closer to a fragment of [8] where we add loop and while constructs. A consequence is that we have to control side effects. Our work is inspired by recent studies on the Implicit Computational Complexity of imperative programs [14, 10]. Contrarily to these seminal works, we work on polynomial algebra instead of matrix algebra. There are at least two reasons for such an approach. Firstly, the use of polynomials gives a clearest intuition and pushes aside a lot of technicalities. Secondly, polynomials give more flexibility in order to deal with method calls.

The paper is organized as follows. After introducing our language and the notion of sup-interpretation of an object-oriented program, we give a criterion, called brotherly criterion, which ensures that each brotherly program computes objects whose size is polynomially bounded by the inputs sizes, even if the program is defined with function calls. To our knowledge, previous works on the implicit computational complexity of imperative programs did not support such a flexibility. Consequently, this criteria seems to be a great improvement on the study of the complexity of imperative programs.

2 Object-oriented Programs

2.1 Syntax of programs

We consider object-oriented programs. Basically a program is composed by three sets of disjoint symbols \mathcal{X} , \mathcal{P} and \mathcal{F} and a set $\mathbf{Class} \subseteq \mathcal{F}$. The set \mathcal{X} represents the set of attributes. Throughout the following paper, we use capital letters X, Y, Z, \dots for attributes. The set \mathcal{P} is the set of parameters which are passed as arguments of a method. The symbols of \mathbf{Class} are the class identifiers. They provide distinct data encodings such as the unary encoding, using the identifier \mathbf{S} for a class having one attribute and the identifier ϵ for a class without any attribute, or the binary encoding, using the class identifiers $\mathbf{1}$ and $\mathbf{0}$. Each function symbol $\mathbf{f} \in \mathcal{F}$ must be defined by one method of some class. A class $\mathbf{C} \in \mathbf{Class}$ is composed by attribute and method declarations, including a particular constructor method, which are described by the following grammar:

Attributes	$\ni A$	$::= \text{var } X; \mid \text{var } X; A$
Expressions	$\ni e$	$::= x \mid X \mid X.f(e_1, \dots, e_n) \mid \text{new } C(e_1, \dots, e_n)$
Commands	$\ni \text{Cm}$	$::= \text{skip} \mid X := e \mid \text{Cm}_1; \text{Cm}_2 \mid \text{loop } X \{ \text{Cm} \}$ $\mid \text{if } e \text{ then } \text{Cm}_1 \text{ else } \text{Cm}_2 \mid \text{while } e \{ \text{Cm} \}$
Methods	$\ni M$	$::= f(x_1, \dots, x_n) \{ \text{Cm}; \text{return } X; \}$
Class	$\ni C$	$::= \text{Class } C \{ A; \text{Cons}; M_1; \dots; M_n; \}$
	main	$::= \text{Class main} \{ A; \text{Cm} \}$

where $X \in \mathcal{X}$, $x, x_1, \dots, x_n \in \mathcal{P}$, $f \in \mathcal{F}$, $e_1, \dots, e_n \in \text{Expressions}$ and $M_1; \dots; M_n \in \text{Methods}$. The method **Cons** is a special constructor method of the shape $C(x_1, \dots, x_n) \{ X_1 := x_1; \dots; X_n := x_n \}$ which appears in each class $C \in \text{Class}$ whenever the class C has n attributes X_1, \dots, X_n . As a consequence, we have $\text{Class} \subseteq \mathcal{F}$. This particular method can only be used in a command of the shape $X := \text{new } C(e_1, \dots, e_n)$. All attributes appearing in the methods of a given class **Class** must belong to the attributes of this class. Finally, we define the main class **main** to be a special class defined by attributes declarations and commands.

We suppose that the attributes and methods of two distinct classes are pairwise distinct.

For notational convenience, we sometimes refer to \bar{e} as a sequence of expressions e_1, \dots, e_n , whenever n is clear from the context.

Given a program **p**, we define a precedence $\geq_{\mathcal{F}}$ on function symbols of \mathcal{F} . Set $f \geq_{\mathcal{F}} g$ if the method defining f is of the shape $f(\bar{x}) \{ \text{Cm}; \text{return } X; \}$ and the function symbol g appears in **Cm**. Take the reflexive and transitive closure of $\geq_{\mathcal{F}}$, that we also note $\geq_{\mathcal{F}}$. It is not difficult to establish that $\geq_{\mathcal{F}}$ is a preorder. Lastly, we say that $f >_{\mathcal{F}} g$ if $f \geq_{\mathcal{F}} g$ and $g \geq_{\mathcal{F}} f$ does not hold. Intuitively, $f >_{\mathcal{F}} g$ means that f cannot call g . Throughout the following paper, we suppose that for each method of the shape $f(\bar{x}) \{ \text{Cm}; \text{return } X; \}$ and for each function symbol g which occurs in **Cm**, $f >_{\mathcal{F}} g$, i.e. there is no recursive call in the program.

For each expression e of a program, we suppose that function symbols $f \in \mathcal{F} - \text{Class}$ appear only in the outermost position of an expression e . This restriction allows to deal with side effects in a clearest fashion. This is not a severe restriction since every program can be transformed efficiently in an equivalent program which does fit this requirement, by adding new attributes for the intermediate computations. For example, a command of the shape $X := V.f(U.g(X))$ is transformed into $Y := U.g(X); X := V.f(Y)$ with Y a fresh attribute. For simplicity, we suppose, that no function symbol appears in the expression e of the commands **if** e **then** **Cm**₁ **else** **Cm**₂ and **while** e **{Cm}**.

The attribute X is not allowed to occur in the command **Cm** of an iteration of the shape **loop** X **{Cm}**. The attribute X is not allowed to occur in a method of the shape $f(\bar{x}) \{ \text{Cm}' ; \text{return } Y; \}$, if the function symbol f

appears in the command \mathbf{Cm} of an iteration $\text{loop } X \{ \mathbf{Cm} \}$. In other words, the program is not allowed to read and to write the attribute X during the execution of a $\text{loop } X \{ \mathbf{Cm} \}$.

Example 2.1: Here is an example of a program of our language:

```

Class Position {
    var X; var Y;
    Position( $x, y$ ) {  $X := x; Y := y;$  }
    move( $x, y$ ) {  $X := X.add(x); Y := Y.add(y);$  return  $X;$  }
    getX() { skip; return  $X;$  }
}

Class main {
    var  $W$ ; var  $U$ ; var  $V$ ; var  $Z$ ;
     $\mathbf{Cm}_1 : V := \text{new Position}(W, U);$ 
     $\mathbf{Cm}_2 : Z := V.\text{move}(W, W);$ 
     $\mathbf{Cm}_3 : U := V.\text{getX}();$ 
}

```

where $\mathbf{Cm}_1, \dots, \mathbf{Cm}_3$ are labels used to refer to commands and **add** is a method which is not described in the program and which corresponds to the unary or binary addition depending on the kind of defined objects.

2.2 Semantics

The domain of computation is the set of objects defined inductively by:

$$\mathbf{Objects} \ni v ::= b \mid b(v_1, \dots, v_n) \quad b \in \mathbf{Class}$$

Given a **main** class having n attributes X_1, \dots, X_n , an object v_i is stored in each X_i at any time.

A ground substitution σ represents a store which consists in a total mapping from \mathcal{X} to objects in **Objects**. Given a ground substitution σ and an attribute X , the notation $\sigma \{ X := u \}$ means that the object stored in $X\sigma$ is updated to the object u in σ . A parameter substitution β is a total mapping from \mathcal{P} to objects in **Objects**. Given an expression e and a ground substitution σ , we use the notation $\langle e, \sigma \rangle \rightarrow \langle u, \sigma' \rangle$ whether the expression e evaluates to u and the store σ is updated to σ' . We use the notation $\langle \mathbf{Cm}, \sigma \rangle \rightarrow \langle \sigma' \rangle$, if σ is updated to σ' during the execution of the command \mathbf{Cm} . Given a program p of main class **Class main** $\{ A; \mathbf{Cm} \}$ and a store σ , p computes a store σ' defined by $\langle \mathbf{Cm}, \sigma \rangle \rightarrow \langle \sigma' \rangle$.

The operational semantics of our language is inspired by the operational semantics of the java fragment given in [8] and is described in Figure 1.

$$\begin{array}{c}
\frac{}{\langle \diamond, \sigma \rangle \rightarrow \langle \diamond \sigma, \sigma \rangle} \diamond \in \mathcal{X} \cup \mathcal{P} \\
\frac{\langle \bar{e}, \sigma \rangle \rightarrow \langle \bar{u}, \sigma \rangle \quad \mathbf{f}(\bar{x}) \{ \mathbf{Cm} ; \mathbf{return} Y ; \} \quad \exists \beta, \bar{x}\beta = \bar{u} \quad \langle \mathbf{Cm}\beta, \sigma \rangle \rightarrow \langle \sigma' \rangle}{\langle X.\mathbf{f}(\bar{e}), \sigma \rangle \rightarrow \langle Y\sigma', \sigma' \rangle} \\
\frac{\langle \bar{e}, \sigma \rangle \rightarrow \langle \bar{u}, \sigma \rangle}{\langle \mathbf{new} \mathbf{C}(\bar{e}), \sigma \rangle \rightarrow \langle \mathbf{C}(\bar{u}), \sigma \rangle} \mathbf{C} \in \mathbf{Class} \\
\frac{\langle e, \sigma \rangle \rightarrow \langle u, \sigma' \rangle}{\langle X := e \rangle \rightarrow \langle \sigma \{ X := u \} \rangle} \\
\frac{}{\langle \mathbf{skip}, \sigma \rangle \rightarrow \langle \sigma \rangle} \\
\frac{\langle \mathbf{Cm}_1, \sigma \rangle \rightarrow \langle \sigma' \rangle \quad \langle \mathbf{Cm}_2, \sigma' \rangle \rightarrow \langle \sigma'' \rangle}{\langle \mathbf{Cm}_1; \mathbf{Cm}_2, \sigma \rangle \rightarrow \langle \sigma'' \rangle} \\
\frac{\langle e, \sigma \rangle \rightarrow \langle \underline{1}, \sigma \rangle, \langle \underline{0}, \sigma \rangle \text{ or } \langle u, \sigma \rangle}{\langle \mathbf{if} \ e \ \mathbf{then} \ \mathbf{Cm}_1 \ \mathbf{else} \ \mathbf{Cm}_2, \sigma \rangle \rightarrow \langle \mathbf{Cm}_1, \sigma \rangle, \langle \mathbf{Cm}_2, \sigma \rangle \text{ or } \langle \mathbf{skip}, \sigma \rangle} \text{ with } u > \underline{1} \\
\frac{}{\langle \mathbf{loop} \ X_i \{ \mathbf{Cm} \}, \sigma \rangle \rightarrow \langle \mathbf{Cm}^{|v_i|}, \sigma \rangle} \text{ with } \mathbf{Cm}^n = \mathbf{Cm}; \mathbf{Cm}^{n-1} \text{ and } \mathbf{Cm}^0 = \mathbf{skip} \\
\frac{\langle e, \sigma \rangle \rightarrow \langle \underline{1}, \sigma \rangle \text{ or } \langle \underline{1}, \sigma \rangle}{\langle \mathbf{while} \ e \{ \mathbf{Cm} \}, \sigma \rangle \rightarrow \langle \mathbf{Cm}; \mathbf{while} \ X \{ \mathbf{Cm} \}, \sigma \rangle \text{ or } \langle \mathbf{skip}, \sigma \rangle} \text{ with } u \neq \underline{1}
\end{array}$$

Figure 1: Call-by-value semantics

If \mathbf{f} is defined by a method of the shape $\mathbf{f}(\bar{x}) \{ \mathbf{Cm} ; \mathbf{return} Y ; \}$, then the evaluation of $X.\mathbf{f}(\bar{u})$ is performed by first evaluating the body $\mathbf{Cm}\beta$, with β a parameter substitution such that $\bar{x}\beta = \bar{u}$, in the context of the object X and then returning the value stored in the attribute Y of the object X .

The command \mathbf{skip} does nothing. The command $X := e$ assigns the value of e to the attribute X . The command $X := \mathbf{new} \mathbf{C}(e_1, \dots, e_n)$ first evaluates the expressions e_1, \dots, e_n to the objects v_1, \dots, v_n , then, it creates a new object of the class \mathbf{C} by assigning the value $\mathbf{C}(v_1, \dots, v_n)$ to the attribute X . The execution of $\mathbf{Cm}_1; \mathbf{Cm}_2$ corresponds to the sequential execution of \mathbf{Cm}_1 and \mathbf{Cm}_2 . $\mathbf{if} \ b \ \mathbf{then} \ \mathbf{Cm}_1 \ \mathbf{else} \ \mathbf{Cm}_2$ executes the command \mathbf{Cm}_1 , \mathbf{Cm}_2 or \mathbf{skip} depending on whether the expression b is respectively evaluated to an encoding of 1, 0 or another natural number. The size $|v|$ of a value v is defined to be the number of symbols of strictly positive arity in v . The command $\mathbf{loop} \ X \{ \mathbf{Cm} \}$ executes $|v|$ times the command \mathbf{Cm} if v is the value

stored in X , i.e. $X\sigma = v$. Finally the command **while** $b \{ \mathbf{Cm} \}$ is evaluated to $\mathbf{Cm}; \mathbf{while} \ b \{ \mathbf{Cm} \}$ if b is evaluated to an encoding of 1 and to **skip** otherwise.

Example 2.2: Consider the program of Example 2.1. For each objects u, v, w, z such that $\sigma = \{U := u, V := v, W := w, Z := z\}$, we have:

$$\begin{aligned} \langle \mathbf{new} \ \mathbf{Position}(W, U), \sigma \rangle &\rightarrow \langle \mathbf{Position}(w, u), \sigma \rangle \\ \langle V := \mathbf{new} \ \mathbf{Position}(W, U), \sigma \rangle &\rightarrow \langle \sigma \{V := \mathbf{Position}(w, u)\} \rangle \end{aligned}$$

Moreover if $v = \mathbf{Position}(w, u)$ then:

$$\begin{aligned} \langle V.\mathbf{getX}(), \sigma \rangle &\rightarrow \langle w, \sigma \rangle \\ \langle U := V.\mathbf{getX}(), \sigma \rangle &\rightarrow \langle \sigma \{U := w\} \rangle \end{aligned}$$

3 Sup-interpretations and weights

3.1 Assignments

Definition 3.1: Given a class \mathbf{C} having n attributes X_1, \dots, X_n , the assignment I of the class \mathbf{C} is a mapping of domain $\text{dom}(I) \subseteq \mathcal{F}$ which assigns a function $I(\mathbf{f}) : (\mathbb{R}^+)^{m+1} \mapsto \mathbb{R}^+$ to every symbol $\mathbf{f} \in \mathcal{F} - \mathbf{Class}$ of arity m , which corresponds to a method of the class \mathbf{C} , and which assigns a function $I(\mathbf{C}) : (\mathbb{R}^+)^n \mapsto \mathbb{R}^+$ to the constructor method of \mathbf{C} .

Given a program p , the assignment I of p consists in the union of the assignments of each class \mathbf{C} of \mathbf{Class} .

A program assignment I is defined over an expression e if each symbol of \mathcal{F} in e belongs to $\text{dom}(I)$. Suppose that the assignment I is defined over an expression e , The partial assignment of e w.r.t. I , that we note $I^*(e)$ is the canonical extension of the assignment I defined as follows:

1. If \diamond is in $\mathcal{X} \cup \mathcal{P}$, then $I^*(\diamond) = \diamond$
2. If \bar{e} is a sequence of expressions e_1, \dots, e_k , $I^*(\bar{e}) = I^*(e_1), \dots, I^*(e_k)$.
3. If \mathbf{C} is a symbol in \mathbf{Class} of arity m and e_1, \dots, e_m are expressions, then, we have:

$$I^*(\mathbf{new} \ \mathbf{C}(e_1, \dots, e_m)) = I(\mathbf{C})(I^*(e_1), \dots, I^*(e_m))$$

4. If $\mathbf{f} \in \mathcal{F} - \mathbf{Class}$ is a symbol of arity m and e_1, \dots, e_m are expressions, then, we have:

$$I^*(X.\mathbf{f}(e_1, \dots, e_m)) = I(\mathbf{f})(I^*(e_1), \dots, I^*(e_m), X)$$

Notice that the assignment $I^*(e)$ of an expression e with m parameters \bar{x} occurring in a class \mathbf{C} having n attributes denotes a function from $(\mathbb{R}^+)^{n+m} \rightarrow \mathbb{R}^+$. Consequently, we use the notation $I^*(e)(X_1, \dots, X_n, \bar{x})$ when we apply such a function.

Definition 3.2: Let $\mathbf{Max-Poly} \{\mathbb{R}^+\}$ be the set of functions defined to be constant functions in \mathbb{R}^+ , projections, \max , $+$, \times and closed by composition. Given a class with n attributes, an assignment I is said to be polynomial if for every symbol b of $\text{dom}(I)$, $I(b)$ is a function of $\mathbf{Max-Poly} \{\mathbb{R}^+\}$.

Definition 3.3: The assignment of a class symbol $\mathbf{C} \in \mathbf{Class}$ of arity $m > 0$ is *additive* if

$$I(\mathbf{C})(\diamond_1, \dots, \diamond_m) = \sum_{i=1}^m \diamond_i + \alpha_{\mathbf{C}} \text{ where } \alpha_{\mathbf{C}} \geq 1$$

If the assignment of each class symbol of strictly positive arity is additive then the assignment is additive.

Definition 3.4: The size of an expression e is noted $|e|$ and defined by $|e| = 0$ if e is a 0-ary symbol and $|b(e_1, \dots, e_m)| = 1 + \sum_i |e_i|$ if $e = b(e_1, \dots, e_m)$ with $m > 0$.

Lemma 3.1: Given a program \mathbf{p} having an additive assignment I , there is a constant α such that for each object $v \in \mathbf{Objects}$, the following inequality is satisfied:

$$|v| \leq I^*(v) \leq \alpha \times |v|$$

Proof: Define $\alpha = \max_{\mathbf{c} \in \mathbf{C}} (\beta_{\mathbf{c}})$ where $\beta_{\mathbf{c}}$ is taken to be the constant $\alpha_{\mathbf{c}}$ of definition 3.1 if \mathbf{c} is of strictly positive arity and $\beta_{\mathbf{c}}$ is equal to the constant $I^*(\mathbf{c})$ otherwise. The inequalities follow directly by induction on the size of a value.

3.2 Sup-interpretations

Definition 3.5: Given a program \mathbf{p} of main class having n attributes X_1, \dots, X_n , a sup-interpretation is an additive assignment θ of \mathbf{p} which satisfies:

1. The assignment θ is weakly monotonic. That is, for each symbol $b \in \text{dom}(\theta)$, the function $\theta(b)$ satisfies:

$$\forall i, \diamond_i \geq \diamond'_i \Rightarrow \theta(b)(\dots, \diamond_i, \dots) \geq \theta(b)(\dots, \diamond'_i, \dots)$$

2. For each function symbol $\mathbf{f} \in \text{dom}(\theta) - \mathbf{Class}$ of arity m , for each m tuple of objects \bar{v} , and for each store σ if $\langle X_i.\mathbf{f}(\bar{v}), \sigma \rangle \rightarrow \langle v, \sigma' \rangle$ then

$$\theta(\mathbf{f})(\theta^*(\bar{v}), \theta^*(X_i\sigma)) \geq \max(\theta^*(v), \theta^*(X_i\sigma'))$$

Intuitively, the sup-interpretation is a special interpretation of a symbol. Instead of yielding the symbol denotation, a sup-interpretation of a function symbol provides an upper bound on the outputs sizes of the function denoted by the symbol. It is worth noticing that sup-interpretation is a complexity measure in the sense of Blum [4].

Example 3.3: Suppose that the method `add` of Example 2.1 is defined over an encoding of unary numbers using two class constructor symbols \mathbf{S} and ϵ of respective arity 1 and 0. It admits the following additive and polynomial sup-interpretation $\theta(\text{add})(\diamond_1, \diamond_2) = \diamond_1 + \diamond_2$, $\theta(\mathbf{S})(\diamond) = \diamond + 1$ and $\theta(\epsilon) = 0$. Indeed, this function is monotonic. For every unary number $\mathbf{S}^v(\epsilon)$, we let the reader check that $\theta^*(\mathbf{S}^v(\epsilon)) = |\mathbf{S}^v(\epsilon)| = v$. Moreover, for every unary number $\mathbf{S}^v(\epsilon)$ and for every store σ such that $X\sigma = \mathbf{S}^u(\epsilon)$, with $\mathbf{S}^{n+1}(\epsilon) = \mathbf{S}(\mathbf{S}^n(\epsilon))$ and $\mathbf{S}^0(\epsilon) = \epsilon$, if $\langle X.\text{add}(\mathbf{S}^v(\epsilon)), \sigma \rangle \rightarrow \langle \mathbf{S}^{v+u}(\epsilon), \sigma \{X := \mathbf{S}^{v+u}(\epsilon)\} \rangle$, then:

$$\begin{aligned} \theta^*(\text{add})(\theta^*(\mathbf{S}^v(\epsilon)), \theta^*(\mathbf{S}^u(\epsilon))) &= \theta^*(\mathbf{S}^u(\epsilon)) + \theta^*(\mathbf{S}^v(\epsilon)) && \text{By Dfn of } \theta \\ &= u + v && \theta^*(\mathbf{S}^v(\epsilon)) = v \\ &\geq \max(u + v, u) \\ &= \max(\theta^*(\mathbf{S}^{u+v}(\epsilon)), \theta^*(\mathbf{S}^u(\epsilon))) \end{aligned}$$

So that, Condition 2 of Definition 3.2 is checked.

Lemma 3.2: Given a program p of main class having n attributes X_1, \dots, X_n and having a sup-interpretation θ defined over an expression e , then, for each parameter substitution β , $\theta^*(e\beta)$ denotes a function from $(\mathbb{R}^+)^n$ to \mathbb{R}^+ which satisfies:

For each store σ , if $\langle e\beta, \sigma \rangle \rightarrow \langle v, \sigma' \rangle$ then

$$\theta^*(e\beta)(\theta^*(X_1\sigma), \dots, \theta^*(X_n\sigma)) \geq \theta^*(v)$$

Moreover, if $e = X_i.\mathbf{f}(e_1, \dots, e_n)$, we have:

$$\theta^*(e\beta)(\theta^*(X_1\sigma), \dots, \theta^*(X_n\sigma)) \geq \theta^*(X_i\sigma')$$

Example 3.4: Consider the program of Example 2.1. As demonstrated in Example 3.2, $\theta(\text{add})(\diamond_1, \diamond_2) = \diamond_1 + \diamond_2$, $\theta(\mathbf{S})(\diamond) = \diamond + 1$ and $\theta(\epsilon) = 0$ define a sup-interpretation for the method `add`. The method `move` admits the following sup-interpretation $\theta(\text{move})(x, y, \diamond) = x + y + \diamond$ and $\theta(\text{Position})(x, y) = x + y + 1$. Indeed, $\theta(\text{move})$ and $\theta(\text{Position})$ are monotonic and, for each store $\sigma = \{U := u, V := \text{Position}(\mathbf{S}^{u_1}(\epsilon), \mathbf{S}^{u_2}(\epsilon)), W := \mathbf{S}^w(\epsilon), Z := z\}$, we have:

$$\langle V.\text{move}(w, w), \sigma \rangle \rightarrow \langle \mathbf{S}^{u_1+w}(\epsilon), \sigma \{V := \text{Position}(\mathbf{S}^{u_1+w}(\epsilon), \mathbf{S}^{u_2+w}(\epsilon))\} \rangle$$

Since $\theta^*(\mathbf{S}^n(\epsilon)) = n = |\mathbf{S}^n(\epsilon)|$, we have to check Condition 2 of Definition 3.2:

$$\begin{aligned}
& \theta^*(V.\text{move}(W\sigma, W\sigma))(\theta^*(u, \text{Position}(\mathbf{S}^{u_1}(\epsilon), \mathbf{S}^{u_2}(\epsilon)), \mathbf{S}^w(\epsilon), z)) \\
&= \theta(\text{move})(\theta^*(W\sigma), \theta^*(W\sigma), V)(\theta^*(u, \text{Position}(\mathbf{S}^{u_1}(\epsilon), \mathbf{S}^{u_2}(\epsilon)), \mathbf{S}^w(\epsilon), z)) \\
&= \theta(\text{move})(\theta^*(W\sigma), \theta^*(W\sigma), \theta^*(\text{Position}(\mathbf{S}^{u_1}(\epsilon), \mathbf{S}^{u_2}(\epsilon)))) \\
&= \theta^*(W\sigma) + \theta^*(W\sigma) + \theta^*(\text{Position}(\mathbf{S}^{u_1}(\epsilon), \mathbf{S}^{u_2}(\epsilon))) \\
&= \theta^*(\mathbf{S}^w(\epsilon)) + \theta^*(\mathbf{S}^w(\epsilon)) + u_1 + u_2 + 1 \\
&\geq \max(w + u_1, 2 \times w + u_1 + u_2 + 1) \\
&= \max(\theta^*(\mathbf{S}^{u_1+w}(\epsilon)), \theta^*(\text{Position}(\mathbf{S}^{u_1+w}(\epsilon), \mathbf{S}^{u_2+w}(\epsilon))))
\end{aligned}$$

3.3 Weights

Now, we are going to define the notion of weight which allows to control the size of the objects held by the attributes during loop and while iterations. Basically, a weight is a partial mapping over commands. The weights depend strongly on the considered command, so that we have to make the distinction between commands.

For that purpose, define the relation \sqsubseteq over commands by $\text{Cm}_1 \sqsubseteq \text{Cm}$

- if there are Cm_2 and Cm_3 such that $\text{Cm} = \text{Cm}_2; \text{Cm}_1; \text{Cm}_3$,
- $\text{Cm} = \text{if } e \text{ then } \text{Cm}_2 \text{ else } \text{Cm}_3$ and $\text{Cm}_1 \sqsubseteq \text{Cm}_2$ or $\text{Cm}_1 \sqsubseteq \text{Cm}_3$,
- $\text{Cm} = \text{loop } X \{ \text{Cm}_2 \}$ and $\text{Cm}_1 \sqsubseteq \text{Cm}_2$,
- or $\text{Cm} = \text{while } e \{ \text{Cm}_2 \}$ and $\text{Cm}_1 \sqsubseteq \text{Cm}_2$,

and its reflexive and transitive closure, that we also note \sqsubseteq . \sqsubseteq defines a partial ordering over commands. The strict relation \sqsubset is defined by $\text{Cm}_1 \sqsubset \text{Cm}$ if $\text{Cm}_1 \sqsubseteq \text{Cm}$ and $\text{Cm}_1 \neq \text{Cm}$

Definition 3.6: A command Cm is said to be:

- flat if there is no Cm_1 of the shape $\text{Cm}_1 = \text{while } e \{ \text{Cm}_2 \}$ or $\text{Cm}_1 = \text{loop } X \{ \text{Cm}_2 \}$ such that $\text{Cm} \sqsubseteq \text{Cm}_2$.
- minimum if there are no commands Cm_1 and Cm_2 and no expression e such that $\text{Cm} = \text{Cm}_1; \text{Cm}_2$ or $\text{Cm} = \text{if } e \text{ then } \text{Cm}_1 \text{ else } \text{Cm}_2$
- whiled if there is a command $\text{Cm}_1 = \text{while } e \{ \text{Cm}_2 \}$ such that $\text{Cm} \sqsubseteq \text{Cm}_1$ or $\text{Cm}_1 \sqsubseteq \text{Cm}$
- looped if there is a command $\text{Cm}_1 = \text{loop } X \{ \text{Cm}_2 \}$ such that $\text{Cm} \sqsubseteq \text{Cm}_1$ and the command Cm_2 is not whiled.

Definition 3.7: Given a program \mathbf{p} having a main class with n attributes, the weight of a command ω is a partial mapping. It assigns to:

- every flat, minimum and looped command \mathbf{Cm} , a total function $\omega_{\mathbf{Cm}}$ from $(\mathbb{R}^+)^{n+1}$ to \mathbb{R}^+
- every flat, minimum and whiled command \mathbf{Cm} , a total function $\omega_{\mathbf{Cm}}$ from $(\mathbb{R}^+)^n$ to \mathbb{R}^+

which satisfy:

1. $\omega_{\mathbf{Cm}}$ is weakly monotonic $\forall i, \diamond_i \geq \diamond'_i \Rightarrow \omega_{\mathbf{Cm}}(\dots, \diamond_i, \dots) \geq \omega_{\mathbf{Cm}}(\dots, \diamond'_i, \dots)$
2. $\omega_{\mathbf{Cm}}$ has the subterm property $\forall i, \forall \diamond_i \in \mathbb{R}^+ \omega_{\mathbf{Cm}}(\dots, \diamond_i, \dots) \geq \diamond_i$

A weight ω is polynomial if each $\omega_{\mathbf{Cm}}$ is a function of **Max-Poly** $\{\mathbb{R}^+\}$.

4 Criteria to control resources

4.1 Brotherly criterion

The brotherly criterion gives constraints on weights and sup-interpretations in order to bound the size of the objects computed by the program by some polynomial in the size of the inputs.

Definition 4.8: A program having a main class with n attributes X_1, \dots, X_n is **brotherly** if there are a polynomial sup-interpretation and a polynomial weight such that:

1. For every flat, minimum and looped command \mathbf{Cm} of the main class, we have:
 - For every expression of the shape $X_j.f(e_1, \dots, e_m)$ occurring in \mathbf{Cm} :

$$\omega_{\mathbf{Cm}}(T+1, X_1, \dots, X_n) \geq \omega_{\mathbf{Cm}}(T, X_1, \dots, X_{j-1}, \theta^*(e)(\overline{X}), X_{j+1}, \dots, X_n)$$

- For every assignment $X_i := e \sqsubseteq \mathbf{Cm}$, we have:

$$\omega_{\mathbf{Cm}}(T+1, X_1, \dots, X_n) \geq \omega_{\mathbf{Cm}}(T, X_1, \dots, X_{i-1}, \theta^*(e)(\overline{X}), X_{i+1}, \dots, X_n)$$

with T is a fresh variable.

2. For every flat, minimum and whiled command \mathbf{Cm} of the main class, we have:
 - For every expression of the shape $X_j.f(e_1, \dots, e_m)$ occurring in \mathbf{Cm} :

$$\omega_{\mathbf{Cm}}(X_1, \dots, X_n) \geq \omega_{\mathbf{Cm}}(X_1, \dots, X_{j-1}, \theta^*(e)(\overline{X}), X_{j+1}, \dots, X_n)$$

- For every assignment $X_i := e \sqsubseteq \mathbf{Cm}$, we have:

$$\omega_{\mathbf{Cm}}(X_1, \dots, X_n) \geq \omega_{\mathbf{Cm}}(X_1, \dots, X_{i-1}, \theta^*(e)(\overline{X}), X_{i+1}, \dots, X_n)$$

Intuitively, the first condition ensures that the size of the objects held by the attributes remains polynomially bounded. The fresh variable T can be seen as a temporal factor which takes into account the number of iterations allowed in a loop. Such a number is polynomially bounded by the size of the objects held by the attributes. The second condition on whiled commands is very similar, however there is no more temporal factor, since we have no piece of information about the termination of a whiled command.

Theorem 4.1: Given a brotherly program p of main class `Class main {A; Cm}` having n attributes X_1, \dots, X_n , there exists a polynomial P such that for any store σ if $\langle \text{Cm}, \sigma \rangle \rightarrow \langle \sigma' \rangle$ then

$$P(|X_1\sigma|, \dots, |X_n\sigma|) \geq \max_{i=1..n} (|X_i\sigma'|)$$

Example 4.5: The program of example 2.1 is brotherly since it admits a polynomial sup-interpretation θ and it has no looped and whiled command.

Example 4.6: Consider the following program, over unary numbers:

```

Class main {
    var X1;
    var X2;
    var X3;
    loop X1 {
        X3 := X3.add(X2);
    };
}

```

$\text{Cm} = \text{loop } X_1 \{ X_3 := X_3.\text{add}(X_2) \}$ is the only minimum, flat and looped command. Applying the brotherly criterion, we have to find a polynomial weight ω_{Cm} and a polynomial sup-interpretation θ such that:

$$\omega_{\text{Cm}}(T + 1, X_1, X_2, X_3) \geq \omega_{\text{Cm}}(T, X_1, X_2, \theta^*(X_3.\text{add}(X_2)))$$

with T is a fresh variable.

Since $\theta(\text{add})(\diamond_1, \diamond_2) = \diamond_1 + \diamond_2$ is a sup-interpretation of the method `add`, previous inequality is satisfied by taking $\omega_{\text{Cm}}(T, \diamond_1, \diamond_2, \diamond_3) = T \times \diamond_3 + \diamond_1 + \diamond_2$ and the program is brotherly.

References

- [1] R. Amadio, S. Coupet-Grimal, S. Dal-Zilio, and L. Jakubiec. A functional scenario for bytecode verification of resource bounds. In *CSL*, volume 3210 of *LNCS*, pages 265–279. Springer, 2004.
- [2] R. Amadio and S. Dal-Zilio. Resource control for synchronous cooperative threads. In *Concur*, pages 68–82, 2004.

- [3] T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.
- [4] M. Blum. A machine-independent theory of the complexity of recursive functions. *Journal of the Association for Computing Machinery*, 14:322–336, 1967.
- [5] G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. Quasi-interpretation a way to control resources. *Submitted to Theoretical Computer Science*, 2005.
- [6] G. Bonfante, J.-Y. Marion, and R. Péchoux. A characterization of alternating log time by first order functional programs. In *LPAR 2006*, volume 4246 of *LNAI*, pages 90–104, 2006.
- [7] S. Dal-Zilio and R. Gascon. Resource bound certification for a tail-recursive virtual machine. In *APLAS 2005*, volume 3780 of *LNCS*, pages 247–263. Springer-Verlag, 2005.
- [8] S. Drossopoulou and S. Eisenbach. Describing the semantics of Java and proving type soundness. *Formal Syntax and Semantics of Java*, pages 41–82, 1999.
- [9] A. Igarashi, B.C. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.
- [10] L. Kristiansen and N.D. Jones. The flow of data and the complexity of algorithms. *New Computational Paradigms*, 3526:263–274.
- [11] C.S. Lee, N.D. Jones, and A.M. Ben-Amram. The size-change principle for program termination. In *Symposium on Principles of Programming Languages*, volume 28, pages 81–92. ACM press, january 2001.
- [12] J.-Y. Marion and R. Péchoux. Sup-interpretations, a semantic method for static analysis of program resources. *Submitted to TOCL*.
- [13] J.-Y. Marion and R. Péchoux. Resource analysis by sup-interpretation. In *FLOPS 2006*, volume 3945 of *LNCS*, pages 163–176, 2006.
- [14] K.-H. Niggl and H. Wunderlich. Certifying polynomial time and linear/polynomial space for imperative programs. *SIAM Journal on Computing*.